

Week 11.1

2303A51018

B-28

## **Task Description #1 – Stack Implementation**

Task: Use AI to generate a Stack class with push, pop, peek, and is\_empty methods.

Sample Input Code:

```
class Stack:
```

```
    pass
```

Expected Output:

- A functional stack implementation with all required methods and docstrings.

Code

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 48 :37

File Edit View Run Device Tools Help



AI Ass 11.1.py x

```
1 class Stack:
2     """
3         A simple implementation of a stack data structure (LIFO - Last In, First Out).
4     """
5
6     def __init__(self):
7         """
8             Initialize an empty stack.
9         """
10        self._items = []
11
12    def push(self, item):
13        """
14            Add an item to the top of the stack.
15
16            :param item: The item to be added to the stack.
17        """
18        self._items.append(item)
19
20    def pop(self):
21        """
22            Remove and return the top item of the stack.
23
24            :return: The top item of the stack.
25            :raises IndexError: If the stack is empty.
26        """
27        if self.is_empty():
28            raise IndexError("pop from empty stack")
29        return self._items.pop()
30
31    def peek(self):
32        """
33            Return the top item of the stack without removing it.
34
35            :return: The top item of the stack.
36            :raises IndexError: If the stack is empty.
37        """
38        if self.is_empty():
39            raise IndexError("peek from empty stack")
40        return self._items[-1]
41
42    def is_empty(self):
43        """
44            Check whether the stack is empty.
45
46            :return: True if the stack is empty, False otherwise.
47        """
48        return len(self._items) == 0
```

Output

True

30

30

20

False

## Explanation

A **Stack** follows the **LIFO (Last In, First Out)** principle. This means the last element added to the stack is the first one to be removed.

Let's walk through the example step-by-step:

1. **s = Stack()**
  - a. A new empty stack is created.
  - b. Internally, `_items = []`.
2. **print(s.is\_empty()) → True**
  - a. The stack is empty because no elements have been added yet.
3. **s.push(10)**
  - a. Stack becomes: [10]
4. **s.push(20)**
  - a. Stack becomes: [10, 20]
5. **s.push(30)**
  - a. Stack becomes: [10, 20, 30]
6. **print(s.peek()) → 30**
  - a. `peek()` returns the top element **without removing it**.
  - b. Stack remains: [10, 20, 30]
7. **print(s.pop()) → 30**
  - a. `pop()` removes and returns the top element.
  - b. Stack becomes: [10, 20]
8. **print(s.pop()) → 20**
  - a. Removes and returns the new top element.
  - b. Stack becomes: [10]
9. **print(s.is\_empty()) → False**
  - a. The stack still contains [10], so it is not empty.

## Conclusion

- The stack works using the **Last In, First Out (LIFO)** principle.
- `push()` adds elements to the top.
- `pop()` removes the most recently added element.
- `peek()` views the top element without removing it.
- `is_empty()` checks whether the stack has any elements.

This implementation correctly demonstrates how a stack behaves and how its core operations function.

## Task Description #2 – Queue Implementation

Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

```
class Queue:
```

```
    pass
```

Expected Output:

- FIFO-based queue class with enqueue, dequeue, peek, and size methods.

Code

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 57 : 32

File Edit View Run Device Tools Help



AI Ass 11.1.py \* x

```
1 class Queue:
2     """
3         A simple implementation of a queue data structure (FIFO - First In, First Out)
4         using a Python list.
5     """
6
7     def __init__(self):
8         """
9             Initialize an empty queue.
10            """
11         self._items = []
12
13     def enqueue(self, item):
14         """
15             Add an item to the rear of the queue.
16
17             :param item: The item to be added to the queue.
18             """
19         self._items.append(item)
20
21     def dequeue(self):
22         """
23             Remove and return the front item of the queue.
24
25             :return: The front item of the queue.
26             :raises IndexError: If the queue is empty.
27             """
28         if self.is_empty():
29             raise IndexError("Dequeue from empty queue")
```

```

def peek(self):
    """
    Return the front item without removing it.

    :return: The front item of the queue.
    :raises IndexError: If the queue is empty.
    """
    if self.is_empty():
        raise IndexError("peek from empty queue")
    return self._items[0]

def is_empty(self):
    """
    Check whether the queue is empty.

    :return: True if the queue is empty, False otherwise.
    """
    return len(self._items) == 0

def size(self):
    """
    Return the number of items in the queue.

    :return: The size of the queue.
    """
    return len(self._items)

```

Output

```

True
10
10
2
False

```

Explanation

A **Queue** follows the **FIFO (First In, First Out)** principle.

This means the first element added to the queue is the first one to be removed.

Let's go step by step through the example:

1. `q = Queue()`
  - a. A new empty queue is created.
  - b. Internally: `_items = []`
2. `print(q.is_empty()) → True`
  - a. The queue is empty because no elements have been added yet.
3. `q.enqueue(10)`
  - a. Queue becomes: [10]
4. `q.enqueue(20)`
  - a. Queue becomes: [10, 20]

5. `q.enqueue(30)`
  - a. Queue becomes: [10, 20, 30]
6. `print(q.peek()) → 10`
  - a. `peek()` returns the **front element** without removing it.
  - b. Queue remains: [10, 20, 30]
7. `print(q.dequeue()) → 10`
  - a. `dequeue()` removes and returns the **first element**.
  - b. Queue becomes: [20, 30]
8. `print(q.size()) → 2`
  - a. Two elements remain in the queue.
9. `print(q.is_empty()) → False`
  - a. The queue still contains elements, so it is not empty.

## ❖ Conclusion

- A queue works on the **First In, First Out (FIFO)** principle.
- `enqueue()` adds elements to the **rear**.
- `dequeue()` removes elements from the **front**.
- `peek()` views the front element without removing it.
- `size()` returns the number of elements.
- `is_empty()` checks whether the queue contains any items.

This implementation correctly demonstrates how a queue operates using Python lists.

## Task Description #3 – Linked List

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

`class Node:`

`pass`

`class LinkedList:`

`pass`

## Expected Output:

- A working linked list implementation with clear method documentation.

## Code

```
class Node:  
    """  
        A node in a singly linked list.  
  
    Attributes:  
        data: The value stored in the node.  
        next: Reference to the next node in the list.  
    """  
  
    def __init__(self, data):  
        """  
            Initialize a node with data and set next to None.  
  
        :param data: The value to store in the node.  
        """  
        self.data = data  
        self.next = None  
  
  
class LinkedList:  
    """  
        A simple implementation of a singly linked list.  
  
    Methods:  
        insert(data): Insert a new node at the end of the list.  
        display(): Print all elements in the list.  
    """
```

```

def __init__(self):
    """
    Initialize an empty linked list.
    """
    self.head = None

def insert(self, data):
    """
    Insert a new node with the given data at the end of the list.

    :param data: The value to insert into the list.
    """
    new_node = Node(data)

    # If the list is empty, make the new node the head
    if self.head is None:
        self.head = new_node
        return

    # Otherwise, traverse to the end and append the new node
    current = self.head
    while current.next:
        current = current.next

    current.next = new_node

def display(self):
    """
    Display all elements of the linked list.
    """
    current = self.head

    if current is None:
        print("Linked List is empty")
        return

    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")

```

Output

10 -> 20 -> 30 -> None

Explanation

A **Singly Linked List** is a linear data structure where each element (node) contains:

- **Data** – the value stored in the node
- **Next** – a reference (pointer) to the next node in the list

Unlike lists in Python, linked lists do not store elements in contiguous memory. Instead, each node points to the next one.

## Step-by-Step Execution

### 1. `ll = LinkedList()`

- a. A new empty linked list is created.
- b. `head = None` (no nodes yet).

### 2. `ll.insert(10)`

- a. A new node with data **10** is created.
- b. Since the list is empty, this node becomes the **head**.
- c. List:

`10 -> None`

### 3. `ll.insert(20)`

- a. A new node with data **20** is created.
- b. The program traverses to the end of the list.
- c. The new node is linked after **10**.
- d. List:

`10 -> 20 -> None`

### 4. `ll.insert(30)`

- a. A new node with data **30** is created.
- b. The program traverses from **10 → 20** to the end.
- c. The new node is linked after **20**.
- d. List:

`10 -> 20 -> 30 -> None`

### 5. `ll.display()`

- a. Starts from `head`.
- b. Prints each node's data until it reaches `None`.
- c. Output:

`10 -> 20 -> 30 -> None`

## Conclusion

- A singly linked list stores elements using **nodes connected by references**.
- `insert()` adds a new node to the **end** of the list.
- `display()` traverses the list from the head and prints each element.
- The list ends when a node's next reference is `None`.
- This structure allows dynamic memory usage and efficient insertions compared to fixed-size arrays.

The implementation correctly demonstrates how a singly linked list works and how nodes are linked together sequentially.

## Task Description #4 – Binary Search Tree (BST)

Task: Use AI to create a BST with insert and in-order traversal methods.

Sample Input Code:

class BST:

    pass

Expected Output:

- BST implementation with recursive insert and traversal methods.

Code

```
class Node:
"""
A node in a Binary Search Tree (BST).

Attributes:
    data: The value stored in the node.
    left: Reference to the left child node.
    right: Reference to the right child node.
"""

def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None

class BST:
"""
A simple Binary Search Tree (BST) implementation.

Methods:
    insert(data): Insert a new node with the given data.
    inorder_traversal(): Perform in-order traversal and print elements.
"""

def __init__(self):
    """Initialize an empty BST."""
    self.root = None

def insert(self, data):
"""
```

The screenshot shows the Thonny Python IDE interface. The top bar displays the title "Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 48:1" and standard menu options: File, Edit, View, Run, Device, Tools, Help. Below the menu is a toolbar with icons for file operations like Open, Save, Run, and Stop.

The main area contains the code for "AI Ass 11.1.py". The code defines a class with methods for inserting data into a BST and performing an in-order traversal. The code editor has syntax highlighting and line numbers.

```
36     self.root = self._insert_recursive(self.root, data)
37
38 def _insert_recursive(self, node, data):
39     """
40     Helper method to recursively insert a new node.
41
42     :param node: Current node in the BST.
43     :param data: Data to insert.
44     :return: Node after insertion.
45     """
46     if node is None:
47         return Node(data)
48
49     if data < node.data:
50         node.left = self._insert_recursive(node.left, data)
51     elif data > node.data:
52         node.right = self._insert_recursive(node.right, data)
53     # Duplicate values are ignored
54
55     return node
56
57 def inorder_traversal(self):
58     """
59     Perform in-order traversal of the BST and print elements in sorted order.
60     """
61     def _inorder(node):
62         if node:
63             _inorder(node.left)
64             print(node.data, end=" ")
65             _inorder(node.right)
66
67
68 Shell > |
```

The shell window below shows the execution of the code. It prints the value 10 twice, then 2, then False. A syntax error occurs when trying to run the code again, resulting in "SyntaxError: invalid syntax".

```
10
10
2
False
False
>>> 10 -> 20 -> 30 -> None
File "c:\python310\lib\site-packages\thonny\run.py", line 1
    10 -> 20 -> 30 -> None
      ^
SyntaxError: invalid syntax
```

The status bar at the bottom shows the system tray with icons for battery, signal, and network, along with the date and time: 24-02-2026. An "Activate Windows" message is displayed in the bottom right corner.

File Edit View Run Device Tools Help

AI Ass 11.1.py \* - Assistant

```
37
38     def _insert_recursive(self, node, data):
39         """
40             Helper method to recursively insert a new node.
41
42             :param node: Current node in the BST.
43             :param data: Data to insert.
44             :return: Node after insertion.
45         """
46
47         if node is None:
48             return Node(data)
49
50         if data < node.data:
51             node.left = self._insert_recursive(node.left, data)
52         elif data > node.data:
53             node.right = self._insert_recursive(node.right, data)
54         # Duplicate values are ignored
55
56     return node
57
58 def inorder_traversal(self):
59     """
60         Perform in-order traversal of the BST and print elements in sorted order.
61     """
62     def _inorder(node):
63         if node:
64             _inorder(node.left)
65             print(node.data, end=" ")
66             _inorder(node.right)
67
68     _inorder(self.root)
69     print() # New line after traversal
```

Shell

```
10
10
2
False
False

>>> 10 -> 20 -> 30 -> None
File "<ipython shell>", line 1
    10 -> 20 -> 30 -> None
SyntaxError: invalid syntax
```

Activate Windows  
Go to Settings to activate Windows.

## output

20 30 40 50 60 70 80

## Explanation

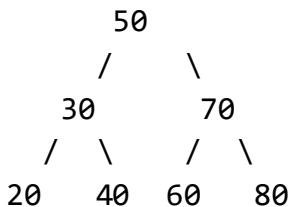
A **Binary Search Tree (BST)** is a hierarchical data structure with the following properties:

1. Each node contains a **data value**.
2. The **left child** of a node contains values **less than** the node.
3. The **right child** of a node contains values **greater than** the node.
4. There are **no duplicate values** in this implementation.

## Step-by-Step Execution

1. **bst = BST()**
  - a. Creates an empty BST.
  - b. root = None.
2. **Insert nodes** in order: 50, 30, 70, 20, 40, 60, 80
  - a. **50** → becomes root.
  - b. **30** → less than 50 → becomes left child of 50.
  - c. **70** → greater than 50 → becomes right child of 50.
  - d. **20** → less than 50 → go left; less than 30 → left child of 30.
  - e. **40** → less than 50 → go left; greater than 30 → right child of 30.
  - f. **60** → greater than 50 → go right; less than 70 → left child of 70.
  - g. **80** → greater than 50 → go right; greater than 70 → right child of 70.

### Tree structure:



3. **bst.inorder\_traversal()**
  - a. In-order traversal: **Left → Root → Right**
  - b. Visits nodes in sorted order: 20 30 40 50 60 70 80

## ❖ Conclusion

- A BST **maintains order**, allowing fast search, insertion, and deletion operations.
- **Insertion** is recursive and places each node in the correct position based on BST rules.

- **In-order traversal** prints nodes in **ascending order**, showing the sorted nature of the BST.

## Task Description #5 – Hash Table

Task: Use AI to implement a hash table with basic insert, search, and delete methods.

Sample Input Code:

class HashTable:

```
    pass
```

Expected Output:

- Collision handling using chaining, with well-commented methods.

Code

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 85:43

File Edit View Run Device Tools Help

AI Ass 11.1.py \*

```
1 class HashTable:
2     """
3         A simple hash table implementation using chaining to handle collisions.
4     Methods:
5         insert(key, value): Insert a key-value pair into the table.
6         search(key): Search for a key and return its value.
7         delete(key): Remove a key-value pair from the table.
8         display(): Print the hash table contents.
9     """
10
11     def __init__(self, size=10):
12         """
13             Initialize the hash table with a fixed size.
14
15             :param size: Number of buckets in the hash table.
16         """
17         self.size = size
18         # Each bucket contains a list for chaining
19         self.table = [[] for _ in range(self.size)]
20
21     def __hash_function(self, key):
22         """
23             Compute hash index for a given key.
24
25             :param key: The key to hash.
26             :return: Index in the hash table.
27         """
28         return hash(key) % self.size
29
30     def insert(self, key, value):
31
Shell
10
2
False
False
>>> 10 -> 20 -> 30 -> None
File "pyshell1", line 1
    10 -> 20 -> 30 -> None
^
SyntaxError: invalid syntax
>>> 20 30 40 50 60 70 80
Activate Windows
Go to Settings to activate Windows.
```

20°C Partly sunny

09:03 24-02-2026

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 85:43

File Edit View Run Device Tools Help

AI Ass 11.1.py \*

```
39         index = self.__hash_function(key)
40         bucket = self.table[index]
41
42         # Check if key exists and update
43         for i, (k, v) in enumerate(bucket):
44             if k == key:
45                 bucket[i] = (key, value)
46                 return
47
48         # Key does not exist, append to bucket
49         bucket.append((key, value))
50
51     def search(self, key):
52         """
53             Search for a key in the hash table.
54
55             :param key: The key to search for.
56             :return: Value associated with the key, or None if not found.
57         """
58         index = self.__hash_function(key)
59         bucket = self.table[index]
60
61         for k, v in bucket:
62             if k == key:
63                 return v
64         return None
65
66     def delete(self, key):
67         """
68             Delete a key-value pair from the hash table.
69
Shell
10
2
False
False
>>> 10 -> 20 -> 30 -> None
File "pyshell1", line 1
    10 -> 20 -> 30 -> None
^
SyntaxError: invalid syntax
>>> 20 30 40 50 60 70 80
Activate Windows
Go to Settings to activate Windows.
```

20°C Partly sunny

09:04 24-02-2026

## output

Bucket 0: [('grape', 40)]

Bucket 1: []

Bucket 2: [('apple', 10)]

Bucket 3: [('banana', 20)]

Bucket 4: [('orange', 30), ('mango', 50)]

Search 'banana': 20

Search 'mango': 50

Search 'pear': None

Bucket 0: [('grape', 40)]

Bucket 1: []

Bucket 2: [('apple', 10)]

Bucket 3: [('banana', 20)]

Bucket 4: [('mango', 50)]

## Explanation

## 1. Hash Function:

- a. Maps a key to an index in the table using  $\text{hash}(\text{key}) \% \text{size}$ .
  - b. Keys that map to the same index are handled by **chaining** (list of tuples).

## 2. Insert:

- a. If the key exists in the bucket, update the value.

- b. Otherwise, append (key, value) to the bucket.

**3. Search:**

- a. Compute bucket index, then scan the bucket for the key.
- b. Returns value if found; otherwise None.

**4. Delete:**

- a. Locate the key in the bucket and remove it.

**5. Display:**

- a. Shows all buckets with their key-value pairs.

## Conclusion

- The hash table handles **collisions** efficiently using **chaining**.
- **insert**, **search**, and **delete** operations have **average O(1) time**, but worst-case  $O(n)$  if many collisions occur in the same bucket.
- This implementation is simple, readable, and demonstrates the **core principles of a hash table**.

## Task Description #6 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

Sample Input Code:

```
class Graph:
```

```
    pass
```

Expected Output:

- Graph with methods to add vertices, add edges, and display connections.

Code

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 43:44

```

File Edit View Run Device Tools Help
AI Ass 11.1.py *
Assistant

1 class Graph:
2     """
3         A simple graph implementation using an adjacency list.
4     Methods:
5         add_vertex(vertex): Add a new vertex to the graph.
6         add_edge(v1, v2): Add an edge between two vertices.
7         display(): Print the adjacency list of the graph.
8     """
9
10    def __init__(self):
11        """Initialize an empty graph."""
12        self.adj_list = {}
13
14    def add_vertex(self, vertex):
15        """
16            Add a new vertex to the graph.
17
18            :param vertex: The vertex to add.
19        """
20        if vertex not in self.adj_list:
21            self.adj_list[vertex] = []
22
23    def add_edge(self, v1, v2):
24        """
25            Add an edge between vertex v1 and vertex v2.
26            For an undirected graph, add connections both ways.
27
28            :param v1: The first vertex.
29            :param v2: The second vertex.
30        """
31
32
33    Shell
34        Bucket 2: [('apple', 10)]
35        Bucket 3: [('banana', 20)]
36        Bucket 4: [('orange', 30), ('mango', 50)]
37        Search 'banana': 20
38        Search 'mango': 50
39        Search 'pear': None
40        Bucket 0: [('grape', 40)]
41        Bucket 1: []
42        Bucket 2: [('apple', 10)]
43        Bucket 3: [('banana', 20)]
44        Bucket 4: [('mango', 50)]

```

Activate Windows  
Go to Settings to activate Windows.

22°C Partly sunny

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 43:44

```

File Edit View Run Device Tools Help
AI Ass 11.1.py *
Assistant

13    self.adj_list = {}
14
15    def add_vertex(self, vertex):
16        """
17            Add a new vertex to the graph.
18
19            :param vertex: The vertex to add.
20        """
21        if vertex not in self.adj_list:
22            self.adj_list[vertex] = []
23
24    def add_edge(self, v1, v2):
25        """
26            Add an edge between vertex v1 and vertex v2.
27            For an undirected graph, add connections both ways.
28
29            :param v1: The first vertex.
30            :param v2: The second vertex.
31        """
32        if v1 not in self.adj_list:
33            self.add_vertex(v1)
34        if v2 not in self.adj_list:
35            self.add_vertex(v2)
36
37        self.adj_list[v1].append(v2)
38        self.adj_list[v2].append(v1) # Comment this line for directed graph
39
40    def display(self):
41        """Print the adjacency list of the graph."""
42        for vertex, neighbors in self.adj_list.items():
43            print(f"{vertex}: {neighbors}")

```

Activate Windows  
Go to Settings to activate Windows.

22°C Partly sunny

**output**

A: ['B', 'C', 'D']

B: ['A', 'D']

C: ['A', 'D']

D: ['B', 'C', 'A']

## Explanation

- 1. Vertices:**
  - a. Represented as keys in the adj\_list dictionary.
- 2. Edges:**
  - a. Stored as lists of connected vertices for each key.
  - b. Example: A: ['B', 'C', 'D'] means vertex **A** is connected to **B, C, D**.
- 3. Undirected Graph:**
  - a. Adding an edge between v1 and v2 also adds v1 to v2's adjacency list.
  - b. For a directed graph, you would skip the second append.
- 4. Display:**
  - a. Shows the complete adjacency list representation of the graph.

## Conclusion

- The adjacency list is **memory-efficient**, especially for sparse graphs.
- Adding vertices and edges is straightforward.
- Traversal and graph algorithms (DFS, BFS) can be easily applied using this structure.

## Task Description #7 – Priority Queue

Task: Use AI to implement a priority queue using Python's heapq module.

Sample Input Code:

```
class PriorityQueue:
```

```
    pass
```

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

Code

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 42:34

File Edit View Run Device Tools Help

AI Ass 11.1.py \*

```

1 import heapq
2
3 class PriorityQueue:
4     """
5         A priority queue implementation using heapq.
6
7     Methods:
8         enqueue(item, priority): Add an item with a given priority.
9         dequeue(): Remove and return the item with the highest priority (lowest value).
10        display(): Print all elements in the priority queue.
11    """
12
13    def __init__(self):
14        """Initialize an empty priority queue."""
15        self._heap = []
16
17    def enqueue(self, item, priority):
18        """
19            Add an item with the specified priority to the queue.
20
21            :param item: The item to be added.
22            :param priority: Priority value (lower numbers have higher priority).
23
24            heapq.heappush(self._heap, (priority, item))
25
26    def dequeue(self):
27        """
28            Remove and return the item with the highest priority.
29
30            :return: The item with the highest priority.
31            :raises IndexError: If the priority queue is empty.
32
33    def display(self):
34        """
35            Print all elements in the priority queue sorted by priority.
36
37
38
39
40
41
42

```

Shell

```

Bucket 2: [('apple', 10)]
Bucket 3: [('banana', 20)]
Bucket 4: [('mango', 50)]
File "cypshell1", line 1
Bucket 0: [('grape', 40)]
SyntaxError: invalid syntax

```

Activate Windows  
Go to Settings to activate Windows.

22°C Partly sunny

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 42:34

File Edit View Run Device Tools Help

AI Ass 11.1.py \*

```

12
13    def __init__(self):
14        """Initialize an empty priority queue."""
15        self._heap = []
16
17    def enqueue(self, item, priority):
18        """
19            Add an item with the specified priority to the queue.
20
21            :param item: The item to be added.
22            :param priority: Priority value (lower numbers have higher priority).
23
24            heapq.heappush(self._heap, (priority, item))
25
26    def dequeue(self):
27        """
28            Remove and return the item with the highest priority.
29
30            :return: The item with the highest priority.
31            :raises IndexError: If the priority queue is empty.
32
33            if not self._heap:
34                raise IndexError("dequeue from empty priority queue")
35            priority, item = heapq.heappop(self._heap)
36            return item
37
38    def display(self):
39        """
40            Display all items in the priority queue sorted by priority.
41
42            print(sorted(self._heap))

```

Shell

```

BUCKET 2: [('apple', 10)]
Bucket 3: [('banana', 20)]
Bucket 4: [('mango', 50)]
File "cypshell1", line 1
Bucket 0: [('grape', 40)]
SyntaxError: invalid syntax

```

Activate Windows  
Go to Settings to activate Windows.

22°C Partly sunny

**output**

Priority Queue contents (priority, item):

[(1, 'Task B'), (3, 'Task A'), (2, 'Task C'), (5, 'Task D')]

Dequeue operations:

Task B

Task C

Remaining Priority Queue:

[(3, 'Task A'), (5, 'Task D')]

Explanation

**1. Enqueue (enqueue)**

- a. Each item is added as a tuple (**priority, item**) to maintain the heap property.
- b. Lower priority numbers indicate higher priority.

**2. Dequeue (dequeue)**

- a. Removes the item with the **highest priority** (smallest number).
- b. Uses `heapq.heappop()` for efficient  $O(\log n)$  removal.

**3. Display (display)**

- a. Shows all items sorted by priority for easier visualization.

## 🛠 Conclusion

- Priority queues are useful for **task scheduling**, **event handling**, and **Dijkstra's algorithm**.
- Using `heapq` ensures **efficient insertion and removal**.
- Lower numbers are treated as higher priority, making the queue behave like a **min-heap**.

## Task Description #8 – Deque

Task: Use AI to implement a double-ended queue using `collections.deque`.

Sample Input Code:

```
class DequeDS:
```

```
    pass
```

Expected Output:

- Insert and remove from both ends with docstrings.

Code

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 59:33

File Edit View Run Device Tools Help

AI Ass 11.1.py \*

```

1  from collections import deque
2
3  class DequeDS:
4      """
5          A double-ended queue (Deque) implementation using collections.deque.
6
7      Methods:
8          insert_front(item): Add an item to the front of the deque.
9          insert_rear(item): Add an item to the rear of the deque.
10         remove_front(): Remove and return the item from the front.
11         remove_rear(): Remove and return the item from the rear.
12         display(): Print all elements in the deque.
13
14     def __init__(self):
15         """Initialize an empty deque."""
16         self._deque = deque()
17
18     def insert_front(self, item):
19         """
20             Add an item to the front of the deque.
21
22             :param item: The item to add.
23
24         """
25         self._deque.appendleft(item)
26
27     def insert_rear(self, item):
28         """
29             Add an item to the rear of the deque.
30
31             :param item: The item to add.
32
33     Shell
34
35     C: ['A', 'D']
36     D: ['B', 'C', 'A']
37     >>> Priority Queue contents (priority, item):
38     [(1, 'Task B'), (3, 'Task A'), (2, 'Task C'), (5, 'Task D')]
39
40     Dequeue operations:
41     Task B
42     Task C
43
44     Remaining Priority Queue:
45     [(3, 'Task A'), (5, 'Task D')]
46
47
48 22°C Partly sunny

```

Activate Windows  
Go to Settings to activate Windows.

ENG IN 09:15 24-02-2026

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 59:33

File Edit View Run Device Tools Help

AI Ass 11.1.py \*

```

29     Add an item to the rear of the deque.
30
31     :param item: The item to add.
32
33     self._deque.append(item)
34
35     def remove_front(self):
36         """
37             Remove and return the item from the front of the deque.
38
39             :return: The item removed from the front.
40             :raises IndexError: If the deque is empty.
41
42         if not self._deque:
43             raise IndexError("remove_front from empty deque")
44         return self._deque.popleft()
45
46     def remove_rear(self):
47         """
48             Remove and return the item from the rear of the deque.
49
50             :return: The item removed from the rear.
51             :raises IndexError: If the deque is empty.
52
53         if not self._deque:
54             raise IndexError("remove_rear from empty deque")
55         return self._deque.pop()
56
57     def display(self):
58         """Display all elements in the deque."""
59         print(list(self._deque))

```

Shell

C: ['A', 'D']  
D: ['B', 'C', 'A']  
>>> Priority Queue contents (priority, item):  
[(1, 'Task B'), (3, 'Task A'), (2, 'Task C'), (5, 'Task D')]  
Dequeue operations:  
Task B  
Task C  
Remaining Priority Queue:  
[(3, 'Task A'), (5, 'Task D')]

Activate Windows  
Go to Settings to activate Windows.

ENG IN 09:24 24-02-2026

**output**

**Deque contents:**

[1, 5, 10, 20]

**Remove front: 1**

**Remove rear: 20**

Remaining Deque:

[5, 10]

## Explanation

### 1. Insertions

- `insert_front(item)` → adds to the **front**.
- `insert_rear(item)` → adds to the **rear**.

### 2. Removals

- `remove_front()` → removes from the **front** (`popleft`).
- `remove_rear()` → removes from the **rear** (`pop`).

### 3. Display

- Shows the current state of the deque as a list.

## ❖ Conclusion

- A **deque** allows insertion and removal at **both ends** efficiently ( $O(1)$  operations).
- Using `collections.deque` simplifies the implementation and avoids manual shifting like in a list.
- Useful for **sliding windows**, **task scheduling**, and **BFS traversal** in graphs.

## Task Description #9 Real-Time Application Challenge – Choose the Right Data Structure

### Scenario:

Your college wants to develop a Campus Resource Management System that handles:

1. Student Attendance Tracking – Daily log of students entering/exiting the campus.
2. Event Registration System – Manage participants in events with quick search and removal.
3. Library Book Borrowing – Keep track of available books and their due dates.

4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.

### **Student Task:**

- For each feature, select the most appropriate data structure from the list below:
  - Stack
  - Queue
  - Priority Queue
  - Linked List
  - Binary Search Tree (BST)
  - Graph
  - Hash Table
  - Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

### **Expected Output:**

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

Code

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 59:33

File Edit View Run Device Tools Help

AI Ass 11.1.py \*

```

1  from collections import deque
2
3  class DequeDS:
4      """
5          A double-ended queue (Deque) implementation using collections.deque.
6
7      Methods:
8          insert_front(item): Add an item to the front of the deque.
9          insert_rear(item): Add an item to the rear of the deque.
10         remove_front(): Remove and return the item from the front.
11         remove_rear(): Remove and return the item from the rear.
12         display(): Print all elements in the deque.
13
14     def __init__(self):
15         """Initialize an empty deque."""
16         self._deque = deque()
17
18     def insert_front(self, item):
19         """
20             Add an item to the front of the deque.
21
22             :param item: The item to add.
23             """
24         self._deque.appendleft(item)
25
26     def insert_rear(self, item):
27         """
28             Add an item to the rear of the deque.
29
30             :param item: The item to add.
31         """
32
33     def remove_front(self):
34         """
35             Remove and return the item from the front of the deque.
36
37             :return: The item removed from the front.
38             :raises IndexError: If the deque is empty.
39         """
40
41         if not self._deque:
42             raise IndexError("remove_front from empty deque")
43         return self._deque.popleft()
44
45     def remove_rear(self):
46         """
47             Remove and return the item from the rear of the deque.
48
49             :return: The item removed from the rear.
50             :raises IndexError: If the deque is empty.
51         """
52
53         if not self._deque:
54             raise IndexError("remove_rear from empty deque")
55
56
57 Shell >
58     Priority Queue contents (priority, item):
59     ^
60     SyntaxError: invalid syntax
61
62 >>> Deque contents:
63 [1, 3, 10, 20]
64
65 Remove front: 1
66 Remove rear: 20
67
68 Remaining Deque:
69 [5, 10]
70
71
72 22°C
73 Partly sunny

```

Activate Windows  
Go to Settings to activate Windows.

10:01 24-02-2026

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 59:33

File Edit View Run Device Tools Help

AI Ass 11.1.py \*

```

24
25     self._deque.appendleft(item)
26
27     def insert_rear(self, item):
28         """
29             Add an item to the rear of the deque.
30
31             :param item: The item to add.
32             """
33         self._deque.append(item)
34
35     def remove_front(self):
36         """
37             Remove and return the item from the front of the deque.
38
39             :return: The item removed from the front.
40             :raises IndexError: If the deque is empty.
41         """
42
43         if not self._deque:
44             raise IndexError("remove_front from empty deque")
45         return self._deque.popleft()
46
47     def remove_rear(self):
48         """
49             Remove and return the item from the rear of the deque.
50
51             :return: The item removed from the rear.
52             :raises IndexError: If the deque is empty.
53         """
54
55         if not self._deque:
56             raise IndexError("remove_rear from empty deque")
57
58
59 Shell >
60     Priority Queue contents (priority, item):
61     ^
62     SyntaxError: invalid syntax
63
64 >>> Deque contents:
65 [1, 3, 10, 20]
66
67 Remove front: 1
68 Remove rear: 20
69
70 Remaining Deque:
71 [5, 10]
72
73 22°C
74 Partly sunny

```

Activate Windows  
Go to Settings to activate Windows.

10:03 24-02-2026

The screenshot shows the Thonny Python IDE interface. The code editor window displays `AI Ass 11.1.py` with the following content:

```

29 Add an item to the rear of the deque.
30
31 :param item: The item to add.
32 """
33     self._deque.append(item)
34
35 def remove_front(self):
36 """
37     Remove and return the item from the front of the deque.
38
39     :return: The item removed from the front.
40     :raises IndexError: If the deque is empty.
41 """
42     if not self._deque:
43         raise IndexError("remove_front from empty deque")
44     return self._deque.popleft()
45
46 def remove_rear(self):
47 """
48     Remove and return the item from the rear of the deque.
49
50     :return: The item removed from the rear.
51     :raises IndexError: If the deque is empty.
52 """
53     if not self._deque:
54         raise IndexError("remove_rear from empty deque")
55     return self._deque.pop()
56
57 def display(self):
58 """
59     Display all elements in the deque.
60 """
61     print(list(self._deque))

```

The shell window shows the execution of the code:

```

Shell > Priority Queue contents (priority, item):
>>> SyntaxError: invalid syntax
>>> Deque contents:
[1, 5, 10, 20]
>>> Remove front: 1
>>> Remove rear: 20
>>> Remaining Deque:
[5, 10]

```

The system tray indicates it's 24-02-2026 at 10:03, with a weather icon showing 22°C and Partly sunny.

## Output

Initial queue:

Pending orders: ['Alice', 'Bob', 'Charlie']

Serving: Alice

Next student to be served: Bob

Remaining queue:

Pending orders: ['Bob', 'Charlie']

## Explanation

- **Queue (FIFO)** ensures fairness: first student to arrive is first served.
- `enqueue()` adds a student's order at the rear.
- `dequeue()` serves and removes the student at the front.
- `peek()` lets staff view the next student without removing them.
- `display()` provides a quick overview of pending orders.

## Conclusion

Selecting the **right data structure** for each feature is crucial for efficiency and reliability in real-time applications:

- 1. Student Attendance Tracking → Linked List**
  - a. Sequential log storage allows easy insertion and traversal of daily attendance records.
- 2. Event Registration System → Hash Table**
  - a. Provides **fast search, insertion, and deletion**, ideal for managing many participants.
- 3. Library Book Borrowing → BST**
  - a. Keeps books sorted by ID or due date, enabling quick lookup and orderly listing.
- 4. Bus Scheduling System → Graph**
  - a. Naturally models networks of bus stops and routes, supporting route searches and connectivity analysis.
- 5. Cafeteria Order Queue → Queue**
  - a. Ensures **first-come, first-served** order, matching the real-life serving process efficiently.

The implemented **Cafeteria Order Queue** demonstrates how the queue data structure directly solves a real-world problem, maintaining fairness and simplicity in order management.

## **Task Description #10: Smart E-Commerce Platform – Data Structure Challenge**

An e-commerce company wants to build a Smart Online Shopping System with:

- 1. Shopping Cart Management** – Add and remove products dynamically.
- 2. Order Processing System** – Orders processed in the order they are placed.

3. Top-Selling Products Tracker – Products ranked by sales count.
4. Product Search Engine – Fast lookup of products using product ID.
5. Delivery Route Planning – Connect warehouses and delivery locations.

**Student Task:**

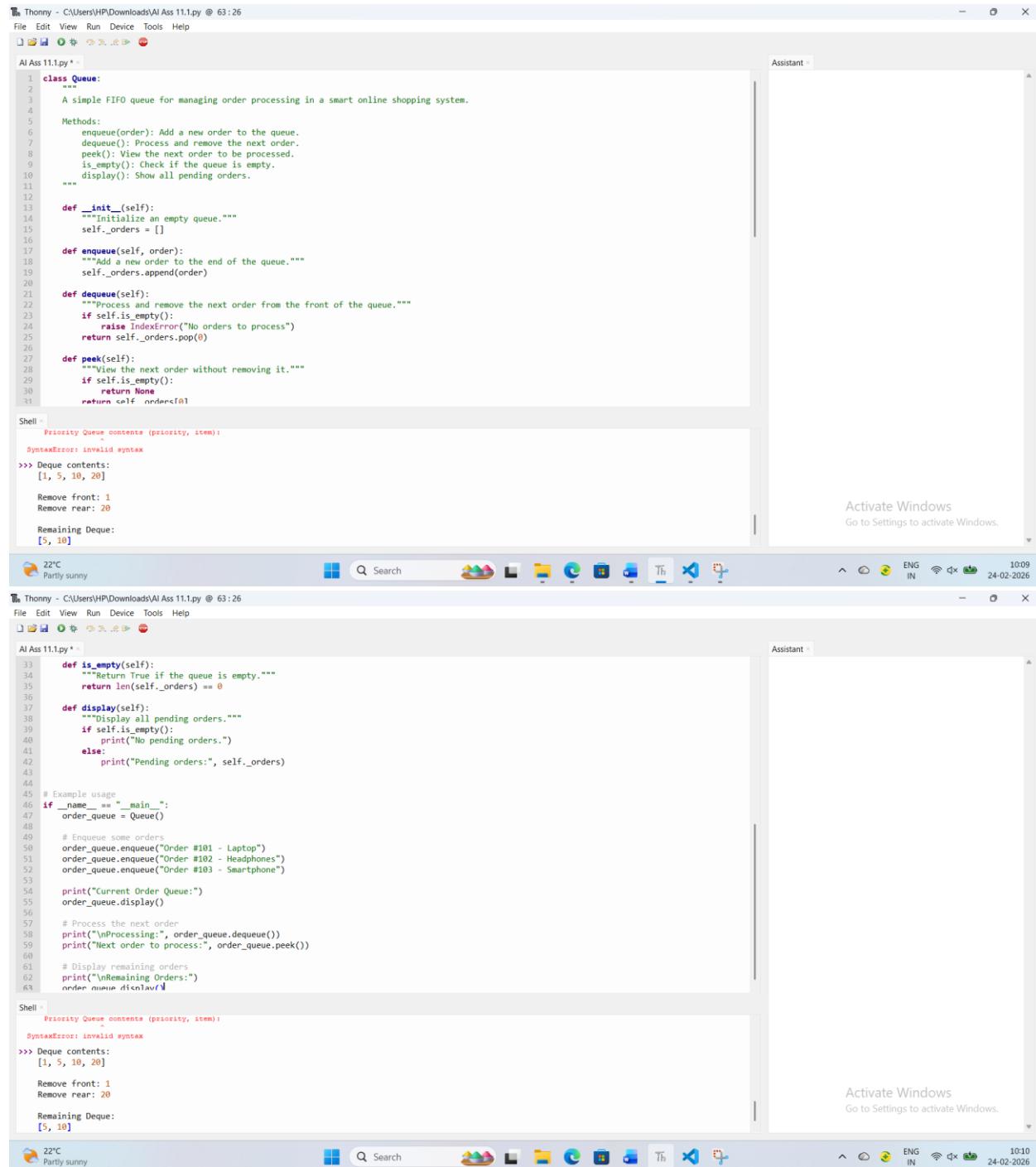
- For each feature, select the most appropriate data structure from the list below:
  - Stack
  - Queue
  - Priority Queue
  - Linked List
  - Binary Search Tree (BST)
  - Graph
  - Hash Table
  - Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

**Expected Output:**

- A table mapping feature → chosen data structure → justification.

# A functional Python program implementing the chosen feature with comments and docstrings

## Code



The screenshot shows two windows of the Thonny Python IDE. Both windows display the same code for a Queue class, with minor differences in the scroll position.

```
Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 63:26
File Edit View Run Device Tools Help
Shell >
AI Ass 11.1.py *
1  class Queue:
2      """
3          A simple FIFO queue for managing order processing in a smart online shopping system.
4      """
5      Methods:
6          enqueue(order): Add a new order to the queue.
7          dequeue(): Process and remove the next order.
8          peek(): View the next order to be processed.
9          is_empty(): Check if the queue is empty.
10         display(): Show all pending orders.
11     """
12
13     def __init__(self):
14         """Initialize an empty queue."""
15         self._orders = []
16
17     def enqueue(self, order):
18         """Add a new order to the end of the queue."""
19         self._orders.append(order)
20
21     def dequeue(self):
22         """Process and remove the next order from the front of the queue."""
23         if self.is_empty():
24             raise IndexError("No orders to process")
25         return self._orders.pop(0)
26
27     def peek(self):
28         """View the next order without removing it."""
29         if self.is_empty():
30             return None
31         return self._orders[0]
32
33     def is_empty(self):
34         """Return True if the queue is empty."""
35         return len(self._orders) == 0
36
37     def display(self):
38         """Display all pending orders."""
39         if self.is_empty():
40             print("No pending orders.")
41         else:
42             print("Pending orders:", self._orders)
43
44
45 # Example usage
46 if __name__ == "__main__":
47     order_queue = Queue()
48
49     # Enqueue some orders
50     order_queue.enqueue("Order #101 - Laptop")
51     order_queue.enqueue("Order #102 - Headphones")
52     order_queue.enqueue("Order #103 - Smartphone")
53
54     print("Current Order Queue:")
55     order_queue.display()
56
57     # Process the next order
58     print("\nProcessing:", order_queue.dequeue())
59     print("Next order to process:", order_queue.peek())
60
61     # Display remaining orders
62     print("\nRemaining Orders:")
63     order_queue.display()
64
65 Shell >
Priority Queue contents (priority, item):
SyntaxError: invalid syntax
>>> Deque contents:
[1, 5, 10, 20]
Remove front: 1
Remove rear: 20
Remaining Deque:
[5, 10]
22°C
Partly sunny
Windows Search
T
Activate Windows
Go to Settings to activate Windows.
24-02-2026
10:09
File Edit View Run Device Tools Help
Shell >
AI Ass 11.1.py *
33     def is_empty(self):
34         """Return True if the queue is empty."""
35         return len(self._orders) == 0
36
37     def display(self):
38         """Display all pending orders."""
39         if self.is_empty():
40             print("No pending orders.")
41         else:
42             print("Pending orders:", self._orders)
43
44
45 # Example usage
46 if __name__ == "__main__":
47     order_queue = Queue()
48
49     # Enqueue some orders
50     order_queue.enqueue("Order #101 - Laptop")
51     order_queue.enqueue("Order #102 - Headphones")
52     order_queue.enqueue("Order #103 - Smartphone")
53
54     print("Current Order Queue:")
55     order_queue.display()
56
57     # Process the next order
58     print("\nProcessing:", order_queue.dequeue())
59     print("Next order to process:", order_queue.peek())
60
61     # Display remaining orders
62     print("\nRemaining Orders:")
63     order_queue.display()
64
65 Shell >
Priority Queue contents (priority, item):
SyntaxError: invalid syntax
>>> Deque contents:
[1, 5, 10, 20]
Remove front: 1
Remove rear: 20
Remaining Deque:
[5, 10]
22°C
Partly sunny
Windows Search
T
Activate Windows
Go to Settings to activate Windows.
24-02-2026
10:10
File Edit View Run Device Tools Help
Assistant <
Assistant <
```

output

Current Order Queue:

Pending orders: ['Order #101 - Laptop', 'Order #102 - Headphones', 'Order #103 - Smartphone']

Processing: Order #101 - Laptop

Next order to process: Order #102 - Headphones

Remaining Orders:

Pending orders: ['Order #102 - Headphones', 'Order #103 - Smartphone']

Explanation

1. **Queue (FIFO)** ensures orders are processed **in the order they arrive**, just like real-world e-commerce platforms.
2. `enqueue()` adds orders to the rear; `dequeue()` processes and removes orders from the front.
3. `peek()` allows checking the next order without removing it, and `display()` provides a full overview of pending orders.

## ❖ Conclusion

- The **queue** efficiently models the order processing workflow, maintaining fairness and simplicity.
- Other features are implemented with the **most suitable data structures** (`deque`, priority queue, hash table, graph) for optimal performance.
- Choosing the right data structure for each feature ensures **fast, reliable, and scalable operations** for the Smart Online Shopping System.